枚举（Enumerations）

> 1.0 翻译：yankuangshi (https://github.com/yankuangshi) 校对：shinyzhu (https://github.com/shinyzhu)
>
> 2.0 翻译+校对：futantan (https://github.com/futantan)
>
> 2.1 翻译：Channe (https://github.com/Channe) 校对：shanks (http://codebuild.me)，
>
> 2.2 翻译+校对：SketchK (https://github.com/SketchK) 2016-05-13

本页内容包含：

*枚举*为一组相关的值定义了一个共同的类型，使你可以在你的代码中以类型安全的方式来使用这些值。

如果你熟悉 C 语言，你会知道在 C 语言中，枚举会为一组整型值分配相关联的名称。Swift 中的枚举更加灵活，不必给每一个枚举成员提供一个值。如果给枚举成员提供一个值（称为“原始”值），则该值的类型可以是字符串，字符，或是一个整型值或浮点数。

此外，枚举成员可以指定任意类型的关联值存储到枚举成员中，就像其他语言中的联合体（unions）和变体（variants）。每一个枚举成员都可以有适当类型的关联值。

在 Swift 中，枚举类型是一等（first-class）类型。它们采用了很多在传统上只被类（class）所支持的特性，例如计算型属性（computed properties），用于提供枚举值的附加信息，实例方法（instance methods），用于提供和枚举值相关联的功能。枚举也可以定义构造函数（initializers）来提供一个初始值；可以在原始实现的基础上扩展它们的功能；还可以遵守协议（protocols）来提供标准的功能。

欲了解更多相关信息，请参见属性（Properties）(./10_Properties.html)，方法（Methods）(./11_Methods.html)，构造过程（Initialization）(./14_Initialization.html)，扩展（Extensions）(./21_Extensions.html)和协议（Protocols）(./22_Protocols.html)。

枚举语法

使用 `enum` 关键词来创建枚举并且把它们的整个定义放在一对大括号内：

```
enum SomeEnumeration {
    // 枚举定义放在这里
}
```

下面是用枚举表示指南针四个方向的例子：

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

枚举中定义的值（如 `North`，`South`，`East` 和 `West`）是这个枚举的*成员值*（或*成员*）。你使用 `case` 关键字来定义一个新的枚举成员值。

> 注意
>
> 与 C 和 Objective-C 不同，Swift 的枚举成员在被创建时不会被赋予一个默认的整型值。在上面的 `CompassPoint` 例子中，`North`，`South`，`East` 和 `West` 不会被隐式地赋值为 `0`，`1`，`2` 和 `3`。相反，这些枚举成员本身就是完备的值，这些值的类型是已经明确定义好的 `CompassPoint` 类型。

多个成员值可以出现在同一行上，用逗号隔开：

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

```
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每个枚举定义了一个全新的类型。像 Swift 中其他类型一样，它们的名字（例如 `CompassPoint` 和 `Planet` ）应该以一个大写字母开头。给枚举类型起一个单数名字而不是复数名字，以便于读起来更加容易理解：

```
var directionToHead = CompassPoint.West
```

`directionToHead` 的类型可以在它被 `CompassPoint` 的某个值初始化时推断出来。一旦 `directionToHead` 被声明为 `CompassPoint` 类型，你可以使用更简短的点语法将其设置为另一个 `CompassPoint` 的值：

```
directionToHead = .East
```

当 `directionToHead` 的类型已知时，再次为其赋值可以省略枚举类型名。在使用具有显式类型的枚举值时，这种写法让代码具有更好的可读性。

使用 Switch 语句匹配枚举值

你可以使用 `switch` 语句匹配单个枚举值：

```
directionToHead = .South
switch directionToHead {
    case .North:
        print("Lots of planets have a north")
    case .South:
        print("Watch out for penguins")
    case .East:
        print("Where the sun rises")
    case .West:
        print("Where the skies are blue")
}
// 输出 "Watch out for penguins”
```

你可以这样理解这段代码：

“判断 `directionToHead` 的值。当它等于 `.North` ，打印 `“Lots of planets have a north”` 。当它等于 `.South` ，打印 `“Watch out for penguins”` 。”

……以此类推。

正如在控制流（Control Flow）(./05_Control_Flow.html)中介绍的那样，在判断一个枚举类型的值时， `switch` 语句必须穷举所有情况。如果忽略了 `.West` 这种情况，上面那段代码将无法通过编译，因为它没有考虑到 `CompassPoint` 的全部成员。强制穷举确保了枚举成员不会被意外遗漏。

当不需要匹配每个枚举成员的时候，你可以提供一个 `default` 分支来涵盖所有未明确处理的枚举成员：

```
let somePlanet = Planet.Earth
switch somePlanet {
case .Earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// 输出 "Mostly harmless”
```

关联值（Associated Values）

上一小节的例子演示了如何定义和分类枚举的成员。你可以为 `Planet.Earth` 设置一个常量或者变量，并在赋值之后查看这个值。然而，有时候能够把其他类型的*关联值*和成员值一起存储起来会很有用。这能让你连同成员值一起存储额外的自定义信息，并且你每次在代码中使用该枚举成员时，还可以修改这个关联值。

你可以定义 Swift 枚举来存储任意类型的关联值，如果需要的话，每个枚举成员的关联值类型可以各不相同。枚举的这种特性跟其他语言中的可识别联合（discriminated unions），标签联合（tagged unions），或者变体（variants）相似。

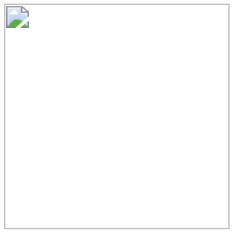
例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有使用 `0` 到 `9` 的数字的 UPC-A 格式的一维条形码。每一个条形码都有一个代表“数字系统”的数字，该数字后接五位代表“厂商代码”的数字，接下来是五位代表“产品代码”的数字。最后一个数字是“检查”位，用来验证代码是否被正确扫描：

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

其他商品上标有 QR 码格式的二维码，它可以使用任何 ISO 8859-1 字符，并且可以编码一个最多拥有 2,953 个字符的字符串：



这便于库存跟踪系统用包含四个整型值的元组存储 UPC-A 码，以及用任意长度的字符串储存 QR 码。

在 Swift 中，使用如下方式定义表示两种商品条形码的枚举：

```
enum Barcode {
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}
```

以上代码可以这么理解：

“定义一个名为 Barcode 的枚举类型，它的一个成员值是具有 (Int，Int，Int，Int) 类型关联值的 UPCA ，另一个成员值是具有 String 类型关联值的 QRCode 。”

这个定义不提供任何 Int 或 String 类型的关联值，它只是定义了，当 Barcode 常量和变量等于 Barcode.UPCA 或 Barcode.QRCode 时，可以存储的关联值的类型。

然后可以使用任意一种条形码类型创建新的条形码，例如：

```
var productBarcode = Barcode.UPCA(8, 85909, 51226, 3)
```

上面的例子创建了一个名为 productBarcode 的变量，并将 Barcode.UPCA 赋值给它，关联的元组值为 (8, 85909, 51226, 3) 。

同一个商品可以被分配一个不同类型的条形码，例如：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

这时，原始的 Barcode.UPCA 和其整数关联值被新的 Barcode.QRCode 和其字符串关联值所替代。 Barcode 类型的常量和变量可以存储一个 .UPCA 或者一个 .QRCode （连同它们的关联值），但是在同一时间只能存储这两个值中的一个。

像先前那样，可以使用一个 switch 语句来检查不同的条形码类型。然而，这一次，关联值可以被提取出来作为 switch 语句的一部分。你可以在 switch 的 case 分支代码中提取每个关联值作为一个常量（用 let 前缀）或者作为一个变量（用 var 前缀）来使用：

```
switch productBarcode {
case .UPCA(let numberSystem, let manufacturer, let product, let check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case .QRCode(let productCode):
    print("QR code: \(productCode).")
}
// 输出 "QR code: ABCDEFGHIJKLMNOP."
```

如果一个枚举成员的所有关联值都被提取为常量，或者都被提取为变量，为了简洁，你可以只在成员名称前标注一个 let 或者 var ：

```
switch productBarcode {
case let .UPCA(numberSystem, manufacturer, product, check):
    print("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case let .QRCode(productCode):
    print("QR code: \(productCode).")
}
// 输出 "QR code: ABCDEFGHIJKLMNOP."
```

原始值（Raw Values）

极客学院
jikexueyuan.com
(http://www.jikexueyuan.com)

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

在关联值小节的条形码例子中，演示了如何声明存储不同类型关联值的枚举成员。作为关联值的替代选择，枚举成员可以被默认值（称为*原始值*）预填充，这些原始值的类型必须相同。

这是一个使用 ASCII 码作为原始值的枚举：

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

枚举类型 `ASCIIControlCharacter` 的原始值类型被定义为 `Character`，并设置了一些比较常见的 ASCII 控制字符。`Character` 的描述详见字符串和字符 (./03_Strings_and_Characters.html)部分。

原始值可以是字符串，字符，或者任意整型值或浮点型值。每个原始值在枚举声明中必须是唯一的。

> 注意
> 原始值和关联值是不同的。原始值是在定义枚举时被预先填充的值，像上述三个 ASCII 码。对于一个特定的枚举成员，它的原始值始终不变。关联值是创建一个基于枚举成员的常量或变量时才设置的值，枚举成员的关联值可以变化。

原始值的隐式赋值（Implicitly Assigned Raw Values）

在使用原始值为整数或者字符串类型的枚举时，不需要显式地为每一个枚举成员设置原始值，Swift 将会自动为你赋值。

例如，当使用整数作为原始值时，隐式赋值的值依次递增 `1` 。如果第一个枚举成员没有设置原始值，其原始值将为 `0` 。

下面的枚举是对之前 `Planet` 这个枚举的一个细化，利用整型的原始值来表示每个行星在太阳系中的顺序：

```
enum Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

在上面的例子中， `Plant.Mercury` 的显式原始值为 `1` ， `Planet.Venus` 的隐式原始值为 `2` ，依次类推。

当使用字符串作为枚举类型的原始值时，每个枚举成员的隐式原始值为该枚举成员的名称。

下面的例子是 `CompassPoint` 枚举的细化，使用字符串类型的原始值来表示各个方向的名称：

```
enum CompassPoint: String {
    case North, South, East, West
}
```

上面例子中， `CompassPoint.South` 拥有隐式原始值 `South` ，依次类推。

使用枚举成员的 `rawValue` 属性可以访问该枚举成员的原始值：

```
let earthsOrder = Planet.Earth.rawValue
// earthsOrder 值为 3

let sunsetDirection = CompassPoint.West.rawValue
// sunsetDirection 值为 "West"
```

使用原始值初始化枚举实例（Initializing from a Raw Value）

如果在定义枚举类型的时候使用了原始值，那么将会自动获得一个初始化方法，这个方法接收一个叫做 `rawValue` 的参数，参数类型即为原始值类型，返回值则是枚举成员或 `nil` 。你可以使用这个初始化方法来创建一个新的枚举实例。

这个例子利用原始值 7 创建了枚举成员 `Uranus` ：

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet 类型为 Planet? 值为 Planet.Uranus
```

然而，并非所有 `Int` 值都可以找到一个匹配的行星。因此，原始值构造器总是返回一个*可选*的枚举成员。在上面的例子中， `possiblePlanet` 是 `Planet?` 类型，或者说"可选的 `Planet` "。

> 注意
> 原始值构造器是一个可失败构造器，因为并不是每一个原始值都有与之对应的枚举成员。更多信息请参见可失败构造器 (../chapter3/05_Declarations.html#failable_initializers)

如果你试图寻找一个位置为 9 的行星，通过原始值构造器返回的可选 `Planet` 值将是 `nil` ：

极客学院
jikexueyuan.com
(http://www.jikexueyuan.com)

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

```
let positionToFind = 9
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
    case .Earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// 输出 "There isn't a planet at position 9
```

这个例子使用了可选绑定（optional binding），试图通过原始值 9 来访问一个行星。 `if let somePlanet = Planet(rawValue: 9)` 语句创建了一个可选 `Planet` ，如果可选 `Planet` 的值存在，就会赋值给 `somePlanet` 。在这个例子中，无法检索到位置为 9 的行星，所以 `else` 分支被执行。

递归枚举（Recursive Enumerations）

*递归枚举（recursive enumeration）*是一种枚举类型，它有一个或多个枚举成员使用该枚举类型的实例作为关联值。使用递归枚举时，编译器会插入一个间接层。你可以在枚举成员前加上 `indirect` 来表示该成员可递归。

例如，下面的例子中，枚举类型存储了简单的算术表达式：

```
enum ArithmeticExpression {
    case Number(Int)
    indirect case Addition(ArithmeticExpression, ArithmeticExpression)
    indirect case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

你也可以在枚举类型开头加上 `indirect` 关键字来表明它的所有成员都是可递归的：

```
indirect enum ArithmeticExpression {
    case Number(Int)
    case Addition(ArithmeticExpression, ArithmeticExpression)
    case Multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

上面定义的枚举类型可以存储三种算术表达式：纯数字、两个表达式相加、两个表达式相乘。枚举成员 `Addition` 和 `Multiplication` 的关联值也是算术表达式——这些关联值使得嵌套表达式成为可能。例如，表达式 `(5 + 4) * 2` ，乘号右边是一个数字，左边则是另一个表达式。因为数据是嵌套的，因而用来存储数据的枚举类型也需要支持这种嵌套——这意味着枚举类型需要支持递归。下面的代码展示了使用 `ArithmeticExpression` 这个递归枚举创建表达式 `(5 + 4) * 2`

```
let five = ArithmeticExpression.Number(5)
let four = ArithmeticExpression.Number(4)
let sum = ArithmeticExpression.Addition(five, four)
let product = ArithmeticExpression.Multiplication(sum, ArithmeticExpression.Number(2))
```

要操作具有递归性质的数据结构，使用递归函数是一种直截了当的方式。例如，下面是一个对算术表达式求值的函数：

```
func evaluate(expression: ArithmeticExpression) -> Int {
    switch expression {
    case .Number(let value):
        return value
    case .Addition(let left, let right):
        return evaluate(left) + evaluate(right)
    case .Multiplication(let left, let right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// 输出 "18"
```

该函数如果遇到纯数字，就直接返回该数字的值。如果遇到的是加法或乘法运算，则分别计算左边表达式和右边表达式的值，然后相加或相乘。

上一篇: 闭包 (/project/swift/chapter2/07_Closures.html)
下一篇: 类和结构体 (/project/swift/chapter2/09_Classes_and_Structures.html)

被顶起来的评论

Via 由 [极客学院 Wiki (http://wiki.jikexueyuan.com)] 提供

馬资懿 (http://weibo.com/emosad)
递归枚举很难理解的样子
(http://weibo.com/emosad)2015年10月19日 回复 顶(2) 转发

四楼iOS住客
灵活灵活，反正灵活，用不用是我们的事，反正可以这样用
1月6日 回复 顶(1) 转发

Lee (http://www.douban.com/people/61905973/)
总觉得枚举搞得这么复杂没有必要~
(http://www.douban.com/people/61905973/)2015年10月16日 回复 顶(1) 转发

31 条评论 2 条新浪微博 最新 最早 最热

浪子潇洞 (http://hujiaweibujidao.github.io/)
如何理解枚举的关联值？枚举的每个case对应的枚举成员到底是什么？
(http://hujiaweibujidao.github.io/)

情况一：只是列出关联值的类型

```
enum Barcode {
case UPCA(Int, Int, Int, Int)
case QRCode(String)
}
```

情况二：列出关联值的类型和名称

```
enum VendingMachineError: ErrorType {
case InvalidSelection //选择无效
case InsufficientFunds(coinsNeeded: Int) //金额不足
case OutOfStock //缺货
}
```

5月9日 回复 顶 转发

Tisoga
回复 NewProgrammer: 可以用来写递归下降解析器
4月6日 回复 顶 转发

永_远_有_多_远 (http://weibo.com/2651440571)
跟c的枚举出入很大，加了不少新的特性，不知道怎么用呢
(http://weibo.com/2651440571)2月23日 回复 顶 转发

如果有如果
学习.....
1月26日 回复 顶 转发

iCode (http://weibo.com/2663764813)
已阅!
(http://weibo.com/2663764813)1月26日 回复 顶 转发

iCode (http://weibo.com/2663764813)
额。先会定义最基本的枚举就好了，还有关联的会定义就好了。
(http://weibo.com/2663764813)1月25日 回复 顶 转发

Ace.
枚举递归的"枚举类型存储了简单的算术表达式："例子编译通不过了，语法有跟新了？ 当前环境版本：swift2.1
1月23日 回复 顶 转发

显卡84du (http://weibo.com/1878289774)
枚举的类型可以是结构或者类吗?
(http://weibo.com/1878289774)1月19日 回复 顶 转发

fair
回复 清风明月的微博小窝74839: 答案在 "使用原始值初始化枚举实例"这一节,没有好好看书
1月8日 回复 顶 转发

四楼iOS住客
灵活灵活，反正灵活，用不用是我们的事，反正可以这样用
1月6日 回复 顶(1) 转发

yss (http://t.qq.com/smzsx1)
这个枚举就是变相的类而已....
(http://t.qq.com/smzsx1)2015年12月30日 回复 顶 转发

lucas

Via 由 [极客学院 Wiki (http://wiki.jikexueyuan.com)] 提供

回复 馬资懿: 其实就是一个递归而已。
2015年12月25日 回复 顶 转发

Belief
个人感觉 有很多类的东西也被分配给了枚举 这样有什么意义 不如像C一样 多简洁
2015年12月23日 回复 顶 转发

Lay
回复 清风明月的微博小窝74839: let name = Planet(rawValue: 4)!
2015年12月21日 回复 顶 转发


G梅果果 (http://weibo.com/2063322140)
感觉分配给递归的功能太多了,用得到么
(http://weibo.com/2063322140)2015年12月18日 回复 顶 转发


这种艺术我很难领悟 (http://weibo.com/talkmore)
跟C#里面的表达式树差不多
(http://weibo.com/talkmore)2015年12月9日 回复 顶 转发


许胜斌 (http://weibo.com/shengbinqiaozhen)
hao
(http://weibo.com/shengbinqiaozhen)2015年12月8日 回复 顶 转发


李东波 (http://t.qq.com/xclidongbo)
虽然理解了,但是,不知道实际的用处何在.
(http://t.qq.com/xclidongbo)2015年11月28日 回复 顶 转发


笑你妹 (http://t.qq.com/ytx2577)
不过, 我连递归都懂不起... 搞得好复杂
(http://t.qq.com/ytx2577)2015年11月26日 回复 顶 转发


笑你妹 (http://t.qq.com/ytx2577)
回复 夏娜: 有的. 显示指定了枚举类型就可以使用rawValue
(http://t.qq.com/ytx2577)2015年11月26日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

「极客学院 Wiki - wiki.jikexueyuan.com」正在使用多说 (http://duoshuo.com)